STATE MANAGEMENT DERSLERİ – AYHAN AKKAYA

**Ders 1 : State Management Nedir?**

Not : Uygulama boyunca değişmeyen yapılar statelessWidget dir. Stateful ise uygulama durumlarını tutar.

1-Öncelikle sayaç uygulamamızda widgetları ayrı bir stateless widget içine alalım.

2-Text yapılarını MyText adında bir sınıfta aşağıdaki gibi tanımlayalım.

class MyText extends StatelessWidget {

  const MyText({super.key});

  @override

  Widget build(BuildContext context) {

    return const Text('You have pushed the button this many times:');

  }

}

3-MyCounterText adında bir sınıfı çıkaralım

class MyCounterText extends StatelessWidget {

  final int sayac;

  const MyCounterText({Key? key, required this.sayac}) : super(key: key);

  @override

  Widget build(BuildContext context) {

    return Text(

      sayac.toString(),

      style: Theme.of(context).textTheme.headline4,

    );

  }

}

4-FloatingActionButton Yapısını da aşağıdaki gibi tanımlayalım

class MyFloatingActionButton extends StatelessWidget {

**VoidCallback onArttir;**

  MyFloatingActionButton({Key? key, required this.onArttir}) : super(key: key);

  @override

  Widget build(BuildContext context) {

    return FloatingActionButton(

**onPressed: () {**

**onArttir();**

**},**

      tooltip: 'Increment',

      child: const Icon(Icons.add),

    );

  }

}

5- body altında aşağıdaki gibi FAB kısmını düzenleyelim.

floatingActionButton: MyFloatingActionButton(

        onArttir: () {

          \_counter++;

          setState(() {});

        },

**Ders 2 : Riverpod State Management Hakkında Teorik Giriş**

1. <https://riverpod.dev/> adresine gidelim.
2. Widget tree yapısını çiz alt ve üst sınıflara gitme hakkında bilgi ver.
3. Flutter\_riverpod paketi projeye eklenir.
4. <https://marketplace.visualstudio.com/items?itemName=robert-brunhage.flutter-riverpod-snippets> adresinden snippet projeye eklenir.
5. <https://riverpod.dev/docs/concepts/providers> adresinden açıklamaları oku.

Providers

Artık Riverpod'u yüklediğimize göre, "Providers" hakkında konuşalım.

Providers, bir Riverpod uygulamasının en önemli parçasıdır. Providers, bir durum parçasını içine alan ve bu durumu dinlemeye izin veren bir nesnedir.

Neden Providers kullanmalısınız?

Bir Providers a bir state parçası sarmak:

Bu duruma birden çok konumda kolayca erişmeyi sağlar. Providers, Singletons, Service Locators, Dependency Injection veya InheritedWidgets gibi kalıpların yerini alır.

Bu durumu başkalarıyla birleştirmeyi kolaylaştırır. Birden fazla nesneyi tek bir nesnede birleştirmek için hiç mücadele ettiniz mi? Bu senaryo doğrudan sağlayıcıların içinde oluşturulmuştur.

Performans optimizasyonlarını etkinleştirir. Widget yeniden oluşturmalarını filtrelemek veya pahalı durum hesaplamalarını önbelleğe almak için; Providers, yalnızca bir durum değişikliğinden etkilenenlerin yeniden hesaplanmasını sağlar.

Uygulamanızın test edilebilirliğini artırır. Providers larda karmaşık kurulum/tearDown adımlarına ihtiyacınız yoktur. Ayrıca, herhangi bir sağlayıcı, bir test sırasında farklı davranacak şekilde geçersiz kılınabilir, bu da çok özel bir davranışın kolayca test edilmesini sağlar.

Günlüğe kaydetme veya yenilemek için çekme gibi gelişmiş özelliklerle kolay entegrasyon sağlar.

Creating a provider

Sağlayıcıların birçok çeşidi vardır, ancak hepsi aynı şekilde çalışır.

En yaygın kullanım, bunları aşağıdaki gibi global sabitler olarak bildirmektir:

*final* myProvider = Provider((ref) {  
 *return* MyValue();  
});

Bu snippet üç bileşenden oluşur:

final myProvider, bir değişkenin bildirimi. Bu değişken, gelecekte provider durumunu okumak için kullanacağımız değişkendir. Providers her zaman nihai olmalıdır.

provider, kullanmaya karar verdiğimiz provider. Provider, tüm providers ların en temelidir. Asla değişmeyen bir nesneyi ortaya çıkarır. Değerin etkileşim şeklini değiştirmek için Provider'ı StreamProvider veya StateNotifierProvider gibi diğer sağlayıcılarla değiştirebiliriz.

Paylaşılan durumu oluşturan bir işlev. Bu işlev her zaman parametre olarak ref adlı bir nesne alır. Bu nesne, diğer sağlayıcıları okumamıza, sağlayıcımızın durumu yok edildiğinde bazı işlemleri gerçekleştirmemize ve çok daha fazlasını sağlar.

Providers a iletilen işlev tarafından döndürülen nesnenin türü, kullanılan provider a bağlıdır. Örneğin, bir provider ın işlevi herhangi bir nesne oluşturabilir. Öte yandan, StreamProvider'ın geri aramasının bir Akış döndürmesi beklenir.

Sınırlama olmaksızın istediğiniz kadar sağlayıcı bildirebilirsiniz. Paket:Provider kullanmanın aksine, Riverpod aynı "tür" durumunu ortaya çıkaran birden çok provider oluşturmaya izin verir:

*final* cityProvider = Provider((ref) => 'London');  
*final* countryProvider = Provider((ref) => 'England');

Sağlayıcıların çalışması için Flutter uygulamalarınızın köküne ProviderScope eklemelisiniz:

*void* main() {  
 runApp(ProviderScope(child: MyApp()));  
}

**Ders 3 : Provider ve StateProvider Kullanımları – Farkları**

1. Main.dart içerisinde main metodunu aşağıdaki gibi değiştirelim.

import 'package:flutter\_riverpod/flutter\_riverpod.dart';

void main()

   {

  runApp(const ProviderScope(child: MyApp()));

}

1. Daha sonra provider\_basics.dart isminde bir dosya oluşturalım ve home kısmına bu sınıfı gönderelim.
2. 3-titleProvider değişkenini aşağıdaki gibi global olarak en üstte oluşturalım.

final titleProvider = Provider<String>((ref)=>'Riverpod Basics');

1. MyHomePage de aşağıdaki gibi sarı alanları silelim.Çünkü artık constructor a gerek kalmadı.

class MyHomePage extends StatefulWidget {

const MyHomePage({Key? key, required this.title}) : super(key: key);

  final String title;

Aşağıdaki alanlar güncellenecek.

 home: const MyHomePage(),

appBar: AppBar(

        title: Text(''),

1. Consumer(Tüketici) yapısını build kısmında ilgili yere yazalım.(1.Yöntem)

appBar: AppBar(

        title: Consumer(builder: ((context, ref, child) {

          var title = ref.watch(titleProvider);

          return Text(title);

        })),

1. En üste global olarak Provider tipinde Butona Basılma Sayısı ekleyelim.

final textProvider = Provider<String>(

  (ref) {

    return 'Butona Basılma Sayısı';

  },

);

1. MyText classında ilgili yere gelinir.StatelessWidget ConsumerWidget olarak değiştirilir.(2.Yöntem)

class MyText extends ConsumerWidget {

  const MyText({Key? key}) : super(key: key);

  @override

  Widget build(BuildContext context, WidgetRef ref) {

    return Text(ref.watch(textProvider));

  }

}

**NOT :** Provider yapısı immutable yani değiştirilmeyecek state içermeyecek yapılar için kullanılır.

**Ders 3\_2 : StateProvider Kısmı**

stateProvider ın provider yapısından farkı değişen içerikler barındırmasıdır.üstelik bu yapılar sayesinde sadece ilgili değişiklikler değişir. Uygulamadaki birçok yer etkilenmez.

1. En üstte global olarak sayacProvider oluşturalım.

final sayacStateProvider = StateProvider<int>(((ref) {

  return 0;

}));

1. Aşağıdaki seçili kısımları silelim.

class \_MyHomePageState extends State<MyHomePage> {

  int \_counter = 0;

MyCounterText(sayac: \_counter),

floatingActionButton: MyFloatingActionButton(

        onArttir: () {

          \_counter++;

          setState(() {});

        },

      ),

class MyCounterText extends StatelessWidget {

  final int sayac;

  const MyCounterText({Key? key, required this.sayac}) : super(key: key);

Widget build(BuildContext context) {

    return Text(

      sayac.toString(),

      style: Theme.of(context).textTheme.headline4,

    );

class MyFloatingActionButton extends StatelessWidget {

  VoidCallback onArttir;

  MyFloatingActionButton({Key? key, required this.onArttir}) : super(key: key);

return FloatingActionButton(

      onPressed: () {

        onArttir();

      },

1. MyCounterText classını aşağıdaki gibi değiştirelim.

class MyCounterText extends ConsumerWidget {

  const MyCounterText({Key? key}) : super(key: key);

  @override

  Widget build(BuildContext context, WidgetRef ref) {

    var sayac = ref.watch(sayacStateProvider);

    return Text(

      sayac.toString(),

      style: Theme.of(context).textTheme.headline4,

    );

  }

}

1. MyFloatingActionButton class ını aşağıdaki şekilde değiştirelim.

lass MyFloatingActionButton extends ConsumerWidget {

  MyFloatingActionButton({Key? key}) : super(key: key);

  @override

  Widget build(BuildContext context, WidgetRef ref) {

    return FloatingActionButton(

      onPressed: () {

        ref.read(sayacStateProvider.state).state++;

      },

      tooltip: 'Increment',

      child: const Icon(Icons.add),

    );

  }

}

Not : Performansı arttırmak için ilgili önerilen yerleri const ekleyelim.

Dilersek StatefullWidget olan ana class ı StatelessWidget yapabiliriz.

**Ders 4 : StateNotifierProvider ile Immutable State Kullanımı 1**

1. Önceki oluşturduğumuz providerları all\_providers.dart isimli bir dosyaya aktaralım.

import 'package:flutter\_riverpod/flutter\_riverpod.dart';

final titleProvider = Provider<String>((ref) => 'Riverpod Basics');

final textProvider = Provider<String>(

  (ref) {

    return 'Butona Basılma Sayısı';

  },

);

final sayacStateProvider = StateProvider<int>(((ref) {

  return 0;

}));

1. Uygulamaların UI ve iş yapan kısımları yani business logic yapıları aynı yerde olmamalı.Diyelim ki firebase üzerinden verileri okuma yazma vs gibi yapılarda farklı bir yapı gerekecek.Bunu bize sağlayacak yapı StateNotifier yapısı olacaktır.
2. lib altında model klasörü oluşturalım.model içinde sayac\_model.dart sınıfı oluşturalım.

class SayacModel {

  final int sayac;

  SayacModel(this.sayac);

}

1. 2.maddede anlatılanlara istinaden sayaç\_manager.dart isminde business logic leri tutan bir yapı oluşturalım.

import 'package:flutter\_riverpod/flutter\_riverpod.dart';

import 'model/sayac\_model.dart';

final sayacNotifierProvider =

    StateNotifierProvider<SayacManager, SayacModel>((ref) {

  return SayacManager();

});

class SayacManager extends StateNotifier<SayacModel> {

  SayacManager() : super(SayacModel(0));

  void arttir() {

    var oankiSayacDegeri = state.sayac;

    state = SayacModel(oankiSayacDegeri + 1);

  }

  void azalt() {

    var oankiSayacDegeri = state.sayac;

    state = SayacModel(oankiSayacDegeri - 1);

  }

}

**Ders 5 : StateNotifierProvider ile Immutable State Kullanımı 2**

1. Aşağıdaki yapıyı all\_providers.dart a ekleyelim.

final sayacNotifierProvider =

    StateNotifierProvider<SayacManager, SayacModel>((ref) {

  return SayacManager();

});

1. lib altında state\_notifier\_kullanimi.dart adında bir dosya oluşturalım.riverpod\_basics.dart dosyasını buraya yapıştıralım.
2. all\_providers.dart içinde title2Provider adında bir yapı oluşturalım.

final title2Provider = Provider<String>((ref) => 'Riverpod State Notifier Kullanimi');

class \_MyHomePageState extends State<MyHomePage> {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Consumer(builder: ((context, ref, child) {

          var title = ref.watch(title2Provider);

          return Text(title);

        })),

1. MyCounterText

class MyCounterText extends ConsumerWidget {

  const MyCounterText({Key? key}) : super(key: key);

  @override

  Widget build(BuildContext context, WidgetRef ref) {

    var sayacDegeri = ref.watch(sayacNotifierProvider);

    return Text(

      sayacDegeri.sayac.toString(),

      style: Theme.of(context).textTheme.headline4,

    );

  }

}

1. MyFloatingActionButton sınıfı

 onPressed: () {

//ref.read(sayacNotifierProvider) bize SayacModeli verir

//arttirma ve azaltma metotları SayacManager sınıfında old.

//bu sınıfı erişmek için .notifier

        ref.read(sayacNotifierProvider.notifier).arttir();

1. all\_providers.dart içinde bu açıklamaları oku

//1. parametre yani SayacManager iş kodlarının, metotların olduğu dosyadır

//Buna erişmek için ref.watch(sayacNotifierProvider.notifier) demek gerekir

//2. parametre yani SayacModel ise bu providerin state'idir.

//Buna erişmek için ref.watch(sayacNotifierProvider) demek gerekir.

//SayacModel sınıfındaki sayaç değeri için

// var sayacDegeri = ref.watch(sayacNotifierProvider).sayacDegeri